Prefix sum array and difference array

From PEGWiki

Given an array of numbers, we can construct a new array by replacing each element by the difference between itself and the previous element, except for the first element, which we simply ignore. This is called the **difference array**, because it contains the first differences of the original array. We will denote the difference array of array A by $D(A)$. For example, the difference array of $A = [9, 2, 6, 3, 1, 5, 0, 7]$ is $D(A) = [2-9, 6-2, 3-6, 1-3, 5-1, 0-5, 7-0]$, or $[-7, 4, -3, -2, 4, -5, 7]$.

We see that the difference array can be computed in linear time from the original array, and is shorter than the original array by one element. Here are implementations in C and Haskell. (Note that the Haskell implementation actually takes a list, but returns an array.)

```
// D must have enough space for n-1 ints
void difference_array(int* A, int n, int* D)
{
    for (int i = 0; i < n-1; i++)
        D[i] = A[i+1] - A[i];
}
```

```
d :: [Int] -> Array Int [Int]
d a = listArray (0, length a - 2) (zipWith (-) (tail a) a)
```

The **prefix sum array** is the opposite of the difference array. Given an array of numbers A and an arbitrary constant c, we first append c onto the front of the array, and then replace each element with the sum of itself and all the elements preceding it. For example, if we start with $A = [9, 2, 6, 3, 1, 5, 0, 7]$, and choose to append the arbitrary value -8 to the front, we obtain $P(-8, A) = [-8, -8+9, -8+9+2, -8+9+2+6, ..., -8+9+2+6+3+1+5+0+7]$, or $[-8, 1, 3, 9, 12, 13, 18, 18, 25]$. Computing the prefix sum array can be done in linear time as well, and the prefix sum array is longer than the original array by one element:

```
// P must have enough space for n+1 ints
void prefix_sum_array(int c, int* A, int n, int* P)
{
    P[0] = c;
    for (int i = 0; i < n; i++)
        P[i+1] = P[i] + A[i];
}
```

```
p :: Int -> [Int] -> Array Int [Int]
p c a = listArray (0, length a) (scanl (+) c a)
```

Note that every array has an infinite number of possible prefix sum arrays, since we can choose whatever value we want for c. For convenience, we usually choose $c = 0$. However, changing the value of c has only the effect of shifting all the elements of $P(c, A)$ by a constant. For example, $P(15, A) = [15, 24, 26, 32, 35, 36, 41, 41, 48]$. However, each element of $P(15, A)$ is exactly 23 more than the corresponding element from $P(-8, A)$.

The functions D and P carry out **reverse processes**. Given an nonempty zero-indexed array A:

1. $D(P(c, A)) = A$ for any c. For example, taking the difference array of $P(-8, A) = [-8, 1, 3, 9, 12, 13, 18, 18, 25]$ gives $[9, 2, 6, 3, 1, 5, 0, 7]$, that is, it restores the original array A.
2. $P(A_0, D(A)) = A$. Thus, taking $D(A) = [-7, 4, -3, -2, 4, -5, 7]$ and $A_0 = 9$ (initial element of A), we have $P(A_0, D(A)) = [9, 2, 6, 3, 1, 5, 0, 7]$, again restoring the original array A.

Contents

Analogy with calculus

These two processes—computing the difference array, and computing a prefix sum array—are the discrete equivalents of differentiation and integration in calculus, which operate on continuous domains. An entry in an array is like the value of a function at a particular point.

- *Reverse processes*:
  - $D(P(c, A)) = A$ for any c. Likewise $\frac{d}{dx}\int_c^x f(t)\,dt = f(x)$ for any c.
  - $P(A_0, D(A)) = A$. Likewise $f(a) + \int_a^x \frac{df}{dt}\,dt = f(x)$.
- *Uniqueness*:
  - A differentiable function $f(x)$ can only have one derivative, $\frac{df}{dx}$. An array A can only have one difference array, $D(A)$.
  - A continuous function $f(x)$ has an infinite number of antiderivatives, $F_c(x) = \int_c^x f(t)\,dt$, where c can be any number in its domain, but they differ only by a constant (their graphs are vertical translations of each other). An array A has an infinite number of prefix arrays $P(c, A)$, but they differ only by a constant (at each entry).
- Given some function $f : [a, b] \to \mathbb{R}$, and the fact that F, an antiderivative of f, satisfies $F(a) = y_0$, we can uniquely reconstruct F. That is, even though f has an infinite number of antiderivatives, we can pin it down to one once we are given the value the antiderivative is supposed to attain on the left edge of f's domain. Likewise, given some array A and the fact that P, a prefix sum array of A, satisfies $P_0 = c$, we can uniquely reconstruct P.
- *Effect on length*:
  - $D(A)$ is shorter than A by one element. Differentiating $f : [a, b] \to \mathbb{R}$ gives a function $f' : (a, b) \to \mathbb{R}$ (shortens the closed interval to an open interval).
  - $P(c, A)$ is longer than A by one element. Integrating $f : (a, b) \to \mathbb{R}$ gives a function $F : [a, b] \to \mathbb{R}$ (lengthens the open interval to a closed interval).

Because of these similarities, we will speak simply of *differentiating* and *integrating* arrays. An array can be differentiated multiple times, but eventually it will shrink to length 0. An array can be integrated any number of times.

Use of prefix sum array

The Fundamental Theorem of Calculus also has an analogue, which is why the prefix sum array is so useful. To compute an integral $\int_a^b f(t)\,dt$, which is like a continuous kind of sum of an infinite number of function values $f(a), f(a+\epsilon), f(a+2\epsilon), ..., f(b)$, we take any antiderivative F, and compute $F(b)-F(a)$. Likewise, to compute the sum of values $A_i, A_{i+1}, A_{i+2}, ..., A_{j-1}$, we will take any prefix array $P(c, A)$ and compute $P_j - P_i$. Notice that just as we can use any antiderivative F because the constant cancels out, we can use any prefix sum array because the initial value cancels out. (Note our use of the left half-open interval.)

Proof: $P_j = c + \sum_{k=0}^{j-1} A_k$ and $P_i = c + \sum_{k=0}^{i-1} A_k$. Subtracting gives $P_j - P_i = \sum_{k=0}^{j-1} A_k - \sum_{k=0}^{i-1} A_k = \sum_{k=i}^{j-1} A_k$ as desired. ■

This is best illustrated *via* example. Let $A = [9, 2, 6, 3, 1, 5, 0, 7]$ as before. Take $P(0, A) = [0, 9, 11, 17, 20, 21, 26, 26, 33]$. Then, suppose we want $A_2 + A_3 + A_4 + A_5 = 6 + 3 + 1 + 5 = 15$. We can compute this by taking $P_6 - P_2 = 26 - 11 = 15$. This is because $P_6 - P_2 = (0 + A_0 + A_1 + A_2 + A_3 + A_4 + A_5) - (0 + A_0 + A_1) = A_2 + A_3 + A_4 + A_5$.

When we use the prefix sum array in this case, we generally use $c = 0$ for convenience (although we are theoretically free to use any value we wish), and speak of the prefix sum array obtained this way as simply *the* prefix sum array.

Example: Counting Subsequences (SPOJ)

Computing the prefix sum array is rarely the most difficult part of a problem. Instead, the prefix sum array is kept on hand because the algorithm to solve the problem makes frequent reference to range sums.

We will consider the problem Counting Subsequences (http://www.spoj.pl/problems/SUBSEQ) from IPSC 2006. Here we are given an array of integers S and asked to find the number of contiguous subsequences of the array that sum to 47.

To solve this, we will first transform array S into its prefix sum array $P(0, S)$. Notice that the sum of each contiguous subsequence $S_i + S_{i+1} + S_{i+2} + ... + S_{j-1}$ corresponds to the difference of two elements of P, that is, $P_j - P_i$. So what we want to find is the number of pairs (i, j) with $P_j - P_i = 47$ and $i < j$. (Note that if $i > j$, we will instead get a subsequence with sum -47.)

However, this is quite easy to do. We sweep through P from left to right, keeping a map of all elements of P we've seen so far, along with their frequencies; and for each element P_j we count the number of times $P_j - 47$ has appeared so far, by looking up that value in our map; this tells us how many contiguous subsequences ending at S_{j-1} have sum 47. And finally, adding the number of contiguous subsequences with sum 47 ending at each entry of S gives the total number of such subsequences in the array. Total time taken is $O(N)$, if we use a hash table implementation of the map.

Use of difference array

The difference array is used to keep track of an array when ranges of said array can be updated all at once. If we have array A and add an increment k to elements $A_i, A_{i+1}, ..., A_{j-1}$, then notice that $D_0, D_1, ..., D_{i-2}$ are not affected; that $D_{i-1} = A_i - A_{i-1}$ is increased by k; that $D_i, D_{i+1}, ..., D_{j-2}$ are not affected; that $D_{j-1} = A_j - A_{j-1}$ is decreased by k; and that $D_j, D_{j+1}, \ldots$ are unaffected. Thus, if we are required to update many ranges of an array in this manner, we should keep track of D rather than A itself, and then integrate at the end to reconstruct A.

Example: Wireless (CCC)

This is the basis of the model solution to Wireless (http://wcipeg.com/problem/ccc09s5) from CCC 2009. We are given a grid of lattice points with up to 30000 rows and up to 1000 columns, and up to 1000 circles, each of which is centered at a lattice point. Each circle has a particular weight associated with it. We want to stand at a lattice point such that the sum of the weights of all the circles covering it is maximized, and to determine how many such lattice points there are.

The most straightforward way to do this is by actually computing the sum of weights of all covering circles at each individual lattice point. However, doing this naively would take up to $30000 \cdot 1000 \cdot 1000 = 3 \cdot 10^{10}$ operations, as we would have to consider each lattice point and each circle and decide whether that circle covers that lattice point.

To solve this, we will treat each column of the lattice as an array A, where entry A_i denotes the sum of weights of all the circles covering the point in row i of that column. Now, consider any of the given circles. Either this circle does not cover any points in this column at all, or the points it covers in this column will all be *consecutive*. (This is equivalent to saying that the intersection of a circle with a line is a line segment.) For example, a circle centered at $(2, 3)$ with radius 5 will cover no lattice points with x-coordinate -10, and as for lattice points with x-coordinate 1, it will cover $(1, -1), (1, 0), (1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (1, 7)$. Thus, the circle covers some first point i and some last point j, and adds some value B to each point that it covers; that is, it adds B to $A_i, A_{i+1}, ..., A_j$. Thus, we will simply maintain the difference array D, noting that the circle adds B to D_{i-1} and takes B away from D_j. We maintain such a difference array for each column; and we perform at most two updates for each circle-column pair; so the total number of updates is bounded by $2 \cdot 1000 \cdot 1000 = 2 \cdot 10^6$. (The detail of how to keep track of the initial element of A is left as an exercise to the reader.) After this, we perform integration on each column and find the maximum values, which requires at most a constant times the total number of lattice points, which is up to $3 \cdot 10^7$.

Multiple dimensions

The prefix sum array and difference array can be extended to multiple dimensions.

Prefix sum array

We start by considering the two-dimensional case. Given an $m \times n$ array A, we will define the prefix sum array $P(A)$ as follows: $P_{i,j} = \sum_{k=0}^{i-1} \sum_{m=0}^{j-1} A_{k,m}$. In other words, the first row of P is all zeroes, the first column of P is all zeroes, and all other elements of P are obtained by adding up some upper-left rectangle of values of A. For example, $P_{2,3} = A_{0,0} + A_{0,1} + A_{0,2} + A_{1,0} + A_{1,1} + A_{1,2}$. Or, more easily expressed, $P_{i,j}$ is the sum of all entries of A with indices in $[0, i) \times [0, j)$ (Cartesian product).

In general, if we are given an n-dimensional array A with dimensions $m_1 \times m_2 \times ... \times m_n$, then the prefix sum array has dimensions $(m_1 + 1) \times (m_2 + 1) \times ... \times (m_n + 1)$, and is defined by $P_{i_1,i_2,...,i_n} = \sum_{k_1=0}^{i_1-1} \sum_{k_2=0}^{i_2-1} ... \sum_{k_n=0}^{i_n-1} A_{k_1,k_2,...,k_n}$; or we can simply say that it is the sum of all entries of A with indices in $[0, i_1) \times [0, i_2) \times ... \times [0, i_n)$.

To compute the prefix sum array in the two-dimensional case, we will scan first down and then right, as suggested by the following C++ implementation:

```
vector<vector<int> > P(vector<vector<int> > A)
{
    int m = A.size();
    int n = A[0].size();
    // Make an m+1 by n+1 array and initialize it with zeroes.
    vector<vector<int> > p(m+1, vector<int>(n+1, 0));
    for (int i = 1; i <= m; i++)
        for (int j = 1; j <= n; j++)
            p[i][j] = p[i-1][j] + A[i-1][j-1];
    for (int i = 1; i <= m; i++)
        for (int j = 1; j <= n; j++)
            p[i][j] += p[i][j-1];
    return p;
}
```

Here, we first make each column of P the prefix sum array of a column of A, then convert each row of P into its prefix sum array to obtain the final values. That is, after the first pair of for loops, $P_{i,j}$ will be equal to $A_{0,j} + A_{1,j} + ... + A_{i-1,j}$, and after the second pair, we will then have $P_{i,j} := P_{i,0}+P_{i,1}+...+P_{i,j-1} = (A_{0,0}+...+A_{i-1,0})+(A_{0,1}+...+A_{i-1,1})+...+(A_{0,j-1}+...+A_{i-1,j-1})$, as desired. The extension of this to more than two dimensions is straightforward.

(Haskell code is not shown, because using functional programming in this case does not make the code nicer.)

After the prefix sum array has been computed, we can use it to add together any rectangle in A, that is, all the elements with their indices in $[x_1, x_2), [y_1, y_2)$. To do so, we first observe that $P_{x_2,y_2} - P_{x_1,y_2}$ gives the sum of all the elements with indices in the box $[x_1, x_2) \times [0, y_2)$. Then we similarly observe that $P_{x_2,y_1} - P_{x_1,y_1}$ corresponds to the box $[x_1, x_2) \times [0, y_1)$. Subtracting gives the box $[x_1, x_2) \times [y_1, y_2)$. This gives a final formula of $P_{x_2,y_2} - P_{x_1,y_2} - P_{x_2,y_1} + P_{x_1,y_1}$.

In the n-dimensional case, to sum the elements of A with indices in the box $[x_{1,0}, x_{1,1}) \times [x_{2,0}, x_{2,1}) \times ... \times [x_{n,0}, x_{n,1})$, we will use the formula $\sum_{k_1=0}^{1} \sum_{k_2=0}^{1} ... \sum_{k_n=0}^{1} (-1)^{n-k_1-k_2-...-k_n} P_{x_{1,k_1},x_{2,k_2},...,x_{n,k_n}}$. We will not state the proof, instead noting that it is a form of the inclusion–exclusion principle.

Example: Diamonds (BOI)

The problem Diamonds (http://wcipeg.com/problem/boi09p6) from BOI '09 is a straightforward application of the prefix sum array in three dimensions. We read in a three-dimensional array A, and then compute its prefix sum array; after doing this, we will be able to determine the sum of any box of the array with indices in the box $[x_1, x_2) \times [y_1, y_2) \times [z_1, z_2)$ as $P_{x_2,y_2,z_2} - P_{x_1,y_2,z_2} - P_{x_2,y_1,z_2} - P_{x_2,y_2,z_1} + P_{x_1,y_1,z_2} + P_{x_1,y_2,z_1} + P_{x_2,y_1,z_1} - P_{x_1,y_1,z_1}$.

Difference array

To use the difference array properly in two dimensions, it is easiest to use a source array A with the property that the first row and first column consist entirely of zeroes. (In multiple dimensions, we will want $A_{i_1,i_2,...,i_n} = 0$ whenever any of the i's are zero.)

We will simply define the difference array D to be the array whose prefix sum array is A. This means in particular that $D_{i,j}$ is the sum of elements of D with indices in the box $[i, i+1) \times [j, j+1)$ (this box contains only the single element $D_{i,j}$). Using the prefix sum array A, we obtain $D_{i,j} = A_{i+1,j+1} - A_{i,j+1} - A_{i+1,j} + A_{i,j}$. In general:

$$D_{i_1,i_2,...,i_n} = \sum_{k_1=0}^{1} \sum_{k_2=0}^{1} ... \sum_{k_n=0}^{1} (-1)^{n-k_1-k_2-...-k_n} A_{i_1+k_1,i_2+k_2,...,i_n+k_n}.$$

Should we actually need to *compute* the difference array, on the other hand, the easiest way to do so is by reversing the computation of the prefix sum array:

```
vector<vector<int> > D(vector<vector<int> > A)
{
    int m = A.size();
    int n = A[0].size();
    // Make an m+1 by n+1 array
    vector<vector<int> > d(m-1, vector<int>(n-1));
    for (int i = 0; i < m-1; i++)
        for (int j = 0; j < n-1; j++)
            d[i][j] = A[i+1][j+1] - A[i+1][j];
    for (int i = m-2; i > 0; i--)
        for (int j = 0; j < n-1; j++)
            d[i][j] -= d[i-1][j];
    return d;
}
```

In this code, we first make each column of D the difference array of the corresponding column of A, and then treat each row as now being the prefix sum array of the final result (the row of the difference array), so we scan backward through it to reconstruct the original array (*i.e.*, take the difference array of the row). (We have to do this backward so that we won't overwrite a value we still need later.) The extension to multiple dimensions is straightforward; we simply have to walk backward over every additional dimension as well.

Now we consider what happens when all elements of A with coordinates in the box given by $[r_1, r_2) \times [c_1, c_2)$ are incremented by k. If we take the difference array of each column of A now, as in the function `D` defined above, we see that for each column, we will have to add k to entry $r_1 - 1$ and subtract it from $r_2 - 1$ (as in the one-dimensional case). Now if we take the difference array of each row of what we've just obtained, we notice that in row number $r_1 - 1$, we've added k to every element in columns in $[c_1, c_2)$ in the previous step, and in row number $r_2 - 1$, we've subtracted k to every element in the same column range, so in the end the effect is to add k to elements D_{r_1-1,c_1-1} and D_{r_2-1,c_2-1}, and to subtract k from elements D_{r_2-1,c_1-1} and D_{r_1-1,c_2-1}.

In the general case, when adding k to all elements of A with indices in the box $[x_{1,0}, x_{1,1}) \times [x_{2,0}, x_{2,1}) \times ... \times [x_{n,0}, x_{n,1})$, a total of 2^n elements of D need to be updated. In particular, element $D_{x_{1,j_1}-1,x_{2,j_2}-1,...,x_{n,j_n}-1}$ (where each of the j's can be either 0 or 1, giving 2^n possibilities in total) is incremented by $(-1)^{j_1+j_2+...+j_n}k$. That is, if we consider an n-dimensional array to be an n-dimensional hypercube of numbers, then the elements to be updated lie on the corners of an n-dimensional hypercube; we 2-color the vertices black and white (meaning two adjacent vertices always have opposite colours), with the lowest corner (corresponding to indices $[x_{1,0} - 1, x_{2,0} - 1, ..., x_{n,0} - 1]$) white; and each white vertex receiving $+k$ and each black vertex $-k$. One can attempt to visualize the effect this has on the prefix sum array in three dimensions, and become convinced that it makes sense in n dimensions. Each element in the prefix sum array $A_{i_1,i_2,...,i_n}$ is the sum of all the elements in some box of the difference array with its lowest corner at the origin, $[0, i_1) \times [0, i_2) \times ... \times [0, i_n)$. If the highest corner actually lies within the hypercube, that is, $x_{1,0} \le i_1 < x_{1,1}, x_{2,0} \le i_2 < x_{2,1}, ..., x_{n,0} \le i_n < x_{n,1}$, then this box is only going to contain the low corner $D_{x_{1,0}-1,x_{2,0}-1,...,x_{n,0}-1}$, which has increased by k; thus, this entry in A has increased by k as well, and this is true of all elements that lie within the hypercube, $[x_{1,0}, x_{1,1}) \times [x_{2,0}, x_{2,1}) \times ... \times [x_{n,0}, x_{n,1})$. If any of the i's are less than the lower bound of the corresponding x, then our box doesn't hit any of the vertices of the hypercube at all, so all these elements are unaffected; and if instead any one of them goes over the upper bound, then our box passes in through one hyperface and out through another, which means that corresponding vertices on the low and high face will either both be hit or both not be hit, and each pair cancels itself out, giving again no change outside the hypercube.

Example: The Cake is a Dessert

The Cake is a Dessert (http://wcipeg.com/problem/cake) from the Waterloo–Woburn 2011 Mock CCC is a straightforward application of the difference array in two dimensions.

A word on the dynamic case

The dynamic version of the problem that the prefix sum array is intended to solve requires us to be able to carry out an intermixed sequence of operations, where each operation either changes an element of A or asks us to determine the sum $A_0 + A_1 + ... + A_{i-1}$ for some i. That is, we need to be able to compute entries of the prefix sum array of an array that is changing (dynamic). If we can do this, then we can do range sum queries easily on a dynamic array, simply by taking the difference of prefix sums as in the static case. This is a bit trickier to do, but is most easily and efficiently accomplished using the binary indexed tree data structure, which is specifically designed to solve this problem and can be updated in $O(\log(m_1)\log(m_2)...\log(m_n))$ time, where n is the number of dimensions and each m is a dimension. Each prefix sum query also takes that amount of time, but we need to perform 2^n of these to find a box sum using the formula given previously, so the running time of the query is asymptotically 2^n times greater than that of the update. In the most commonly encountered one-dimensional case, both query and update are simply $O(\log m)$.

The dynamic version of the problem that the difference array is intended to solve is analogous: we want to be able to increment entire ranges of an array, but at any point we also want to be able to determine any element of the array. Here, we simply use a binary indexed tree for the difference array, and compute prefix sums whenever we want to access an element. Here, it is the update that is slower by a factor of 2^n than the query (because we have to update 2^n entries at a time, but only query one.)

On the other hand, if we combine the two, that is, we wish to be able to increment ranges of some array A as well as query prefix sums of A in an intermixed sequence, matters become more complicated. Notice that in the static case we would just track the difference array of A and then integrate twice at the end. In the dynamic case, however, we need to use a segment tree with lazy propagation. This will give us $O(2^n \log(m_1)\log(m_2)...\log(m_n))$ time on both the query and the update, but is somewhat more complex and slower than the binary indexed tree on the invisible constant factor.

Retrieved from "http://wcipeg.com/wiki/index.php?title=Prefix_sum_array_and_difference_array&oldid=1772"

Category: Pages needing diagrams